Type Inference in Context

Adam Gundry * Conor McBride
University of Strathclyde, Glasgow
{adam.gundry,conor.mcbride}@cis.strath.ac.uk

James McKinna †
Radboud University, Nijmegen
james.mckinna@cs.ru.nl

Abstract

We consider the problems of first-order unification and type inference from a general perspective on problem-solving, namely that of information increase in the problem context. This leads to a powerful technique for implementing type inference algorithms. We describe a unification algorithm and illustrate the technique for the familiar Hindley-Milner type system, but it can be applied to more advanced type systems. The algorithms depend on *well-founded* contexts: type variable bindings and type-schemes for terms may depend only on earlier bindings. We ensure that unification yields a most general unifier, and that type inference yields principal types, by advancing definitions earlier in the context only when necessary.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Algorithms, Theory

1. Introduction

Algorithm $\mathcal{W}$ is a well-known type inference algorithm for the Hindley-Milner (HM) system [Damas and Milner 1982; Milner 1978], based on Robinson's Unification Algorithm [1965]. The system consists of simply-typed λ-calculus with 'let-expressions' for polymorphic definitions. For example,

$$\text{let } i := \lambda x.x \text{ in } i\ i$$

is well-typed: i is given a polymorphic type, which is instantiated in two different ways. The syntax of types is

$$\tau ::= \alpha \mid \tau \triangleright \tau.$$

For simplicity, the function arrow $\triangleright$ is our only type constructor. We let α and β range over type variables and τ and υ over types.

Most presentations of Algorithm $\mathcal{W}$ have treated the underlying unification algorithm as a 'black box', but by considering both together we can give a more elegant type inference algorithm. In particular, the generalisation step (used when inferring the type of a let-expression) becomes straightforward (Section 9).

* Supported by the Microsoft Research PhD Scholarship Programme.
† Supported by the NWO cluster 'DIAMANT'.

MSFP'10, September 25, 2010, Baltimore, Maryland, USA.

1.1 Motivating context

Why revisit Algorithm $\mathcal{W}$? As a first step towards a longer-term goal: explaining how to elaborate high-level *dependently typed* programs into fully explicit calculi. Just as $\mathcal{W}$ specialises polymorphic type schemes, elaboration involves inferring *implicit arguments* by solving constraints, but with fewer algorithmic guarantees. Pragmatically, we need to account for stepwise progress in problem solving from states of partial knowledge. We seek local correctness criteria for type inference that guarantee global correctness.

In contrast to other presentations of unification and HM type inference, our algorithm is based on contexts carrying variable definitions as well as declarations. This avoids the need to represent substitutions explicitly. (We use them to reason about the system.)

This paper has been a long time brewing. Its origins lie in a constraint engine cannibalised by McBride from an implementation of Miller's 'mixed prefix' unification [1992], mutating the quantifier prefix into a context. McBride's thesis [1999] gives an early account of using typing contexts to represent the state of an interactive construction system, 'holes' in programs and proofs being specially designated variables. Contexts carry an information order: increase of information preserves typing and equality judgments; proof tactics are admissible context validity rules which increase information; unification is specified as a tactic which increases information to make an equation hold, but its implementation is not discussed. This view of construction underpinned the implementation of Epigram [McBride and McKinna 2004a] and informed Norell's Agda implementation [2007]. It is high time we began to explain how it works and perhaps to understand it.

We are grateful to an anonymous referee for pointing out the work of Dunfield [2009] on polymorphism in a bidirectional type system. Dunfield uses well-founded contexts that contain existential type variables (amongst other things). These variables can be solved, and there is an informal notion of information increase between input and output contexts. However, our concerns are different: whilst Dunfield elaborates a particular approach to bidirectional polymorphic checking to a larger class of type theories, here we pursue a methodological understanding of the problem-solving strategy in Hindley-Milner type inference.

This paper is literate Haskell, with full source code available at `http://personal.cis.strath.ac.uk/~adam/type-inference/`.

1.2 The occurs check

Testing whether a variable occurs in a term is used by both Robinson unification and Algorithm $\mathcal{W}$. In unification, the check is (usually) necessary to ensure termination, let alone correctness: the equation $\alpha \equiv \alpha \triangleright \beta$ has no (finite) solution because the right-hand side depends on the left, so it does not make a good definition.

In Algorithm $\mathcal{W}$, the occurs check is used to discover type dependencies just in time for generalisation. When inferring the type of the let-expression let $x := e'$ in e, the type of e' must first be inferred, then quantified over 'generic' type variables, i.e. those involved with e' but not the enclosing bindings. The rule in question, as presented by Clément et al. [1986], is:

$$\frac{A \vdash e' : \tau' \qquad A_x \cup \{x : \sigma\} \vdash e : \tau}{A \vdash \text{let } x := e' \text{ in } e : \tau}\ \sigma = gen(A, \tau')$$

$$gen(A, \tau) = \begin{cases} \forall \vec{\alpha_i}.\tau & (FV(\tau) \setminus FV(A) = \{\alpha_1, \cdots, \alpha_n\}) \\ \tau & (FV(\tau) \setminus FV(A) = \emptyset) \end{cases}.$$

The context A is an unordered set of type scheme bindings, with A_x denoting 'A minus any x binding ': such contexts do not reflect lexical scope, so shadowing requires deletion and reinsertion.

The 'let' rule is the only real complexity in Algorithm $\mathcal{W}$, and as Milner [1978] wrote, "the reader may still feel that our rules are arbitrarily chosen and only partly supported by intuition." The rules are well-chosen indeed; perhaps we can recover the intuition.

In both cases, the occurs check is used to detect dependencies between variables. Type variables are traditionally left floating in space and given meaning by substitution, but by exposing structure we can manage definitions and dependencies as we go. Recording type variables in the context is natural when dealing with dependent types, as there is no distinction between type and term variables, but it also works well in the simply-typed setting.

2. Unification over a context

We begin by revisiting unification for type expressions containing free variables. Let us equip ourselves to address the problem—solving equations—by explaining which types are considered equal, raising the question of which things a given context admits as types, and hence, which contexts make sense in the first place.

$$\boxed{\Gamma \vdash \textbf{valid}}$$

$$\frac{}{\mathcal{E} \vdash \textbf{valid}} \qquad \frac{\Gamma \vdash \textbf{valid}}{\Gamma, \alpha{:=}? \vdash \textbf{valid}}\ \alpha \notin \Gamma$$

$$\frac{\Gamma \vdash \textbf{valid} \quad \Gamma \vdash \tau\ \textbf{type}}{\Gamma, \alpha{:=}\tau \vdash \textbf{valid}}\ \alpha \notin \Gamma$$

$$\boxed{\Gamma \vdash \tau\ \textbf{type}}$$

$$\frac{\Gamma, \alpha{:=}\_, \Gamma' \vdash \textbf{valid}}{\Gamma, \alpha{:=}\_, \Gamma' \vdash \alpha\ \textbf{type}} \qquad \frac{\Gamma \vdash \tau\ \textbf{type} \quad \Gamma \vdash \upsilon\ \textbf{type}}{\Gamma \vdash \tau \triangleright \upsilon\ \textbf{type}}.$$

$$\boxed{\Gamma \vdash \tau \equiv \upsilon}$$

$$\frac{\Gamma, \alpha{:=}\tau, \Gamma' \vdash \textbf{valid}}{\Gamma, \alpha{:=}\tau, \Gamma' \vdash \alpha \equiv \tau} \qquad \frac{\Gamma \vdash \tau\ \textbf{type}}{\Gamma \vdash \tau \equiv \tau} \qquad \frac{\Gamma \vdash \upsilon \equiv \tau}{\Gamma \vdash \tau \equiv \upsilon}$$

$$\frac{\Gamma \vdash \tau_0 \equiv \upsilon_0 \quad \Gamma \vdash \tau_1 \equiv \upsilon_1}{\Gamma \vdash \tau_0 \triangleright \tau_1 \equiv \upsilon_0 \triangleright \upsilon_1} \qquad \frac{\Gamma \vdash \tau_0 \equiv \tau_1 \quad \Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \tau_0 \equiv \tau_2}$$

Figure 1. Rules for validity, types and type equivalence

The rules in Figure 1 define a context as a left-to-right list of type variables, each of which may be declared *unknown* (written $\alpha{:=}?$) or *defined* (written $\alpha{:=}\tau$). A context is *valid* if the type τ in every definition makes sense in its preceding context. For example, the context $\alpha{:=}?, \beta{:=}?, \gamma{:=}\alpha \triangleright \beta$ is valid, while $\alpha{:=}\beta, \beta{:=}?$ is not, because β is not in scope for the definition of α. This topological sorting of the dependency graph means that entries on the right are harder to depend on, and correspondingly easier to generalise, just by discharging them as hypotheses in the usual way.

Definitions in the context induce a nontrivial equational theory on types, starting with $\alpha \equiv \tau$ for every definition $\alpha{:=}\tau$ in the context, then taking the congruence closure. Unification is the problem of making variable definitions (thus increasing information) in order to make an equation hold. The idea is to decompose constraints on the syntactic structure of types until we reach variables, then move through the context and update it to solve the equation.

For example, we might start in context $\alpha{:=}?, \beta{:=}?, \gamma := \alpha \triangleright \beta$ aiming to solve the equation $\beta \triangleright \alpha \equiv \gamma$. It suffices to define $\beta{:=}\alpha$, giving as final judgment $\alpha{:=}?, \beta{:=}\alpha, \gamma{:=}\alpha \triangleright \beta \vdash \beta \triangleright \alpha \equiv \gamma$.

A context represents a substitution in 'triangular form' [Baader and Snyder 2001], which can be applied on demand. As we proceed with the development, the context structure will evolve to hold a variety of information about variables of all sorts and some control markers, managing the generalisation process.

2.1 Implementation of unification

Figure 2 renders our unification algorithm in Haskell. Algorithm $\mathcal{W}$ has been formally verified in Isabelle/HOL by Naraschewski and Nipkow [1999], using a counter for fresh name generation and a monad to propagate failure; we use similar techniques here.

Figure 2(a) implements types as a functor parameterised by a type of variable names; for simplicity, we use integers. We compute free type variables using the typeclass FTV with membership function $(\in)$. The typeclass instances are derived using Foldable, thanks to a language extension in GHC 6.12 [GHC Team 2009].

Figure 2(b) defines context entries, contexts and suffixes. The types Bwd and Fwd, whose definitions are omitted, are backwards and forwards lists with $\mathcal{E}$ for the empty list and :< and :> for snoc and cons respectively. Lists are monoids under concatenation $(\oplus)$; the 'fish' operator $(<><)$ appends a suffix to a context. We later extend Entry to handle term variables, so this definition is incomplete.

Figure 2(c) defines the Contextual monad of computations which mutate the context or fail. The TyName component is the next fresh name to use; it is an implementation detail not mentioned in the typing rules. The fresh function generates a fresh name and appends its declaration to the context. Our choice of TyName makes it easy to choose a name fresh with respect to a Context.

Figure 2(d) implements onTop, which delivers the typical access pattern for contexts, locally bringing the top variable declaration into focus and working over the remainder. The local operation f, passed as an argument, may restore the previous entry, or it may return a context extension (containing at least as much information as the entry that has been removed) with which to replace it.

Figure 2(e) gives the actual implementations of unification and solution. Unification proceeds structurally over types. If it reaches a pair of variables, it examines the context, using onTop to pick out a variable declaration to consider. Depending on the variables, it then either succeeds, restoring the old entry or replacing it with a new one, or continues with an updated constraint.

The solve function is called to unify a variable with a non-variable type. It works similarly to unify on variables, but must accumulate a list of the type's dependencies and push them left through the context. It also performs the occurs check and calls the monadic fail if an illegal occurrence (leading to an infinite type) is detected.

```
data Ty a = V a | Ty a ▷ Ty a
  deriving (Functor, Foldable)

type TyName = Integer
type Type   = Ty TyName

class FTV a where
  (∈) :: TyName → a → Bool

instance FTV TyName where
  (∈) = (≡)

instance (Foldable t, FTV a) ⇒ FTV (t a) where
  α ∈ t = any (α ∈) t
```

(a) Types, type variables, occurs check

```
data TyDecl  = !Type | ?
data TyEntry = TyName := TyDecl

instance FTV TyEntry where
  α ∈ (_:=!τ) = α ∈ τ
  α ∈ (_:=?)  = False

data Entry   = TY TyEntry | ...
type Context = Bwd Entry
type Suffix  = Fwd TyEntry

(<><) :: Context → Suffix → Context
Γ <>< ε          = Γ
Γ <>< (α := d :> Ξ) = Γ :< TY (α := d) <>< Ξ
```

(b) Contexts and suffixes

```
type Contextual = StateT (TyName, Context) Maybe

fresh :: TyDecl → Contextual TyName
fresh d = do (β, Γ) ← get
             put (succ β, Γ :< TY (β := d))
             return β

getContext :: Contextual Context
getContext = gets snd

putContext :: Context → Contextual ()
putContext Γ = do β ← gets fst
                  put (β, Γ)

modifyContext :: (Context → Context) → Contextual ()
modifyContext f = getContext >>= putContext ∘ f
```

(c) Context manipulation monad

```
data Extension = Restore | Replace Suffix

onTop :: (TyEntry → Contextual Extension)
         → Contextual ()
onTop f = do
  Γ :< vD ← getContext
  putContext Γ
  case vD of
    TY αD → do m ← f αD
               case m of
                 Replace Ξ → modifyContext (<>< Ξ)
                 Restore   → modifyContext (:<vD)
    _     → onTop f ≫ modifyContext (:<vD)

restore :: Contextual Extension
restore = return Restore

replace :: Suffix → Contextual Extension
replace = return ∘ Replace
```

(d) Processing the context

```
unify :: Type → Type → Contextual ()
unify (τ0 ▷ τ1) (v0 ▷ v1) = unify τ0 v0 ≫ unify τ1 v1
unify (V α)     (V β)     = onTop $
  λ(γ := d) → case
    (γ ≡ α, γ ≡ β, d ) of
    (True,  True,  _ ) → restore
    (True,  False, ? ) → replace (α:=!(V β) :> ε)
    (False, True,  ? ) → replace (β:=!(V α) :> ε)
    (True,  False, !τ) → unify (V β) τ     ≫ restore
    (False, True,  !τ) → unify (V α) τ     ≫ restore
    (False, False, _ ) → unify (V α) (V β) ≫ restore
unify (V α)     τ         = solve α ε τ
unify τ         (V α)     = solve α ε τ

solve :: TyName → Suffix → Type → Contextual ()
solve α Ξ τ = onTop $
  λ(γ := d) → let occurs = γ ∈ τ ∨ γ ∈ Ξ in case
    (γ ≡ α, occurs, d ) of
    (True,  True,  _ ) → fail "Occurrence detected!"
    (True,  False, ? ) → replace (Ξ ⊕ (α:=!τ :> ε))
    (True,  False, !v) → modifyContext (<>< Ξ)
                         ≫ unify v τ
                         ≫ restore
    (False, True,  _ ) → solve α (γ := d :> Ξ) τ
                         ≫ replace ε
    (False, False, _ ) → solve α Ξ τ
                         ≫ restore
```

(e) Unification

Figure 2. Haskell implementation of unification

As an example, consider the behaviour of the algorithm when unify is called to solve $\alpha \triangleright \beta \equiv \alpha' \triangleright (\gamma \triangleright \gamma)$:

$$
\begin{array}{lllll}
 & \alpha{:=}?, \beta{:=}?, \alpha'{:=}\beta, & \gamma{:=}? & & \text{initially} \\
 & \alpha{:=}?, \beta{:=}?, \alpha'{:=}\beta, & \gamma{:=}?, & [\alpha \equiv \alpha'] & \\
 & \alpha{:=}?, \beta{:=}?, \alpha'{:=}\beta, & [\alpha \equiv \alpha'], & \gamma{:=}? & \\
 & \alpha{:=}?, \beta{:=}?, [\alpha \equiv \beta], & \alpha'{:=}\beta, & \gamma{:=}? & \\
\twoheadrightarrow & \alpha{:=}?, \quad \beta{:=}\alpha, & \alpha'{:=}\beta, & \gamma{:=}? & \alpha \equiv \alpha' \\
 & \alpha{:=}?, \beta{:=}\alpha, \alpha'{:=}\beta, & \gamma{:=}?, & [\beta \equiv \gamma \triangleright \gamma] & \\
 & \alpha{:=}?, \beta{:=}\alpha, \alpha'{:=}\beta, & [\gamma{:=}? \mid \beta \equiv \gamma \triangleright \gamma] & & \\
 & \alpha{:=}?, \beta{:=}\alpha, [\gamma{:=}? \mid \beta \equiv \gamma \triangleright \gamma] & \alpha'{:=}\beta & & \\
 & \alpha{:=}?, [\gamma{:=}? \mid \alpha \equiv \gamma \triangleright \gamma], \beta{:=}\alpha, & \alpha'{:=}\beta & & \\
\twoheadrightarrow & \gamma{:=}?, \quad \alpha{:=}\gamma \triangleright \gamma, & \beta{:=}\alpha, & \alpha'{:=}\beta & \beta \equiv \gamma \triangleright \gamma
\end{array}
$$

The constraint decomposes into two constraints on variables. The first ignores γ, moves past α' by updating the constraint to $\alpha \equiv \beta$, then defines $\beta := \alpha$. The second calls solve, which collects γ in the dependency suffix, ignores α', moves past β by updating the constraint to $\alpha \equiv \gamma \triangleright \gamma$, then defines α after pasting in γ.

3. Modelling statements-in-context

Given this implementation of unification, let us try to understand it. We would like a general picture of 'statements-in-context' that allows us to view unification and type inference in a uniform setting. What is the common structure?

A *context* is a list of *declarations* assigning *properties* to names (in particular, those of type variables). We let Γ, Δ, Θ range over contexts. The empty context is written $\mathcal{E}$. Let $\mathcal{V}_{\mathrm{TY}}$ be a set of type variables and $\mathcal{D}_{\mathrm{TY}}$ the properties assignable to them: the 'unknown' property $:=?$ and 'defined' properties $:=\tau$, one for each type τ.

Later we introduce corresponding definitions for term variables. Where needed we let $K \in \{\mathrm{TY}, \mathrm{TM}\}$ represent an arbitrary sort of variable. We write xD for an arbitrary property, with $x \in \mathcal{V}_K$ and $D \in \mathcal{D}_K$. The set of variables of Γ with sort K is written $\mathcal{V}_K(\Gamma)$.

We will build a set $\mathcal{S}$ of *statements*, assertions that can be judged in contexts. For now, the grammar of statements will be

$$S ::= \mathbf{valid} \mid \tau\ \mathbf{type} \mid \tau \equiv \upsilon \mid S \wedge S,$$

meaning (respectively) that the context is valid, τ is a type, the types τ and υ are equivalent, and both conjuncts hold.

A statement has zero or more *parameters*, each of which has an associated *sanity condition*, i.e. a statement whose truth is presupposed for the original statement to make sense. The **valid** statement has no parameter and hence no sanity conditions. In τ **type**, the parameter τ has sanity condition **valid**. The type equivalence statement $\tau \equiv \upsilon$ has two parameters, with sanity conditions τ **type** and υ **type** respectively. Finally, $S \wedge S'$ has parameters (and sanity conditions) taken from S and S'.

Each declaration in the context causes some statement to hold. We maintain a map $[\![\cdot]\!]_K : \mathcal{V}_K \times \mathcal{D}_K \to \mathcal{S}$ from declarations to statements. (Typically we will omit the subscript K.) The idea is that $[\![xD]\!]$ is the statement that holds by virtue of the declaration xD in the context. For type variables, we define

$$
\begin{array}{rcl}
[\![\alpha{:=}?]\!] & \mapsto & \alpha\ \mathbf{type} \\
[\![\alpha{:=}\tau]\!] & \mapsto & \alpha\ \mathbf{type} \wedge \alpha \equiv \tau.
\end{array}
$$

We can inspect the context in derivations using the inference rule

$$\textsc{Lookup}\ \frac{xD \in \Gamma}{\Gamma \Vdash [\![xD]\!]}.$$

Note the different turnstile in the conclusion of this rule. We write the *normal judgment* $\Gamma \vdash S$ to mean that the declarations in Γ support the statement S. We write the *neutral judgment* $\Gamma \Vdash S$ to mean that S follows directly from a fact in Γ. Neutral judgments capture exactly the valid appeals to declarations in the context, just as 'neutral terms' in λ-calculus are applied variables, the 'atoms' of terms. Such appeals to the context are the atoms of derivations.

The LOOKUP rule is our only means to extract information from the context, so we omit contextual plumbing (almost) everywhere else. For example, embedding neutral judgments into the normal:

$$\textsc{Neutral}\ \frac{\Vdash S}{\vdash S}.$$

3.1 Validity of contexts

It is not enough for contexts to be lists of declarations: they must be well-founded, that is, each declaration should make sense in *its* context. A context is *valid* if it declares each name at most once, and the assigned property D is meaningful in the preceding context. Rules for the context validity statement **valid** are given in Figure 3.

$$\boxed{\Gamma \vdash \mathbf{valid}}$$

$$\frac{}{\mathcal{E} \vdash \mathbf{valid}} \qquad \frac{\Gamma \vdash \mathbf{valid} \quad \Gamma \vdash \mathbf{ok}_K D}{\Gamma, xD \vdash \mathbf{valid}}\ x \in \mathcal{V}_K \setminus \mathcal{V}_K(\Gamma)$$

Figure 3. Rules for context validity

The map $\mathbf{ok}_K : \mathcal{D}_K \to \mathcal{S}$, for each $K \in \mathcal{K}$, associates the statement of being meaningful, $\mathbf{ok}_K D$, to each D. For types:

$$
\begin{array}{rcl}
\mathbf{ok}_{\mathrm{TY}}(:=?) & \mapsto & \mathbf{valid} \\
\mathbf{ok}_{\mathrm{TY}}(:=\tau) & \mapsto & \tau\ \mathbf{type}
\end{array}
$$

Henceforth we assume that all contexts treated are valid, and ensure we only construct valid ones. We typically ignore freshness issues, as our simple counter implementation suffices for most purposes.

3.2 Rules for establishing statements

Figure 4 gives rules for establishing statements other than **valid**. We deduce that variables are types by lookup in the context, but we need a structural rule for the $\triangleright$ type constructor.

$$\boxed{\tau\ \mathbf{type}} \qquad \boxed{\tau \equiv \upsilon} \qquad \boxed{S \wedge S'}$$

$$\frac{\tau\ \mathbf{type} \quad \upsilon\ \mathbf{type}}{\tau \triangleright \upsilon\ \mathbf{type}} \qquad \frac{\tau\ \mathbf{type}}{\tau \equiv \tau} \qquad \frac{\upsilon \equiv \tau}{\tau \equiv \upsilon} \qquad \frac{\tau_0 \equiv \tau_1 \quad \tau_1 \equiv \tau_2}{\tau_0 \equiv \tau_2}$$

$$\frac{\tau_0 \equiv \upsilon_0 \quad \tau_1 \equiv \upsilon_1}{\tau_0 \triangleright \tau_1 \equiv \upsilon_0 \triangleright \upsilon_1} \qquad \frac{S \quad S'}{S \wedge S'} \qquad \frac{\Vdash S \wedge S'}{\Vdash S} \qquad \frac{\Vdash S \wedge S'}{\Vdash S'}$$

Figure 4. Rules for types, equivalence and conjunction

Statement conjunction $S \wedge S'$ allows us to package multiple facts about a single variable, with a normal introduction rule (pairing) and neutral elimination rules (projections). This is but one instance of a general pattern: we add *normal* introduction rules for composite forms, but supply eliminators only for statements ultimately resting on (composite) hypotheses, obtained by LOOKUP. This forces derivations to be cut-free, facilitating reasoning by induction on derivations. Adding the corresponding projections for *normal* judgments would hamper us in obtaining a syntax-directed rule system. In any case, we shall ensure that the corresponding elimination rules are admissible, as is clearly the case for conjunction.

4. An information order for contexts

The transition from $\alpha :=?$ to $\alpha := \tau$ intuitively cannot falsify any existing equations. More generally, if we rely on the context to tell us what we may deduce about variables, then making contexts more informative must preserve derivability of judgments.

Let Γ and Δ be contexts. A *substitution from* Γ *to* Δ is a map δ from $\mathcal{V}_{\mathrm{TY}}(\Gamma)$ to $\{\tau \mid \Delta \vdash \tau\ \mathbf{type}\}$. We could also substitute for term variables, and give a more general definition, but we omit this for simplicity. Substitutions act on types and statements as usual. Composition of substitutions θ, δ is given by $(\theta \cdot \delta)(\alpha) = \theta(\delta\alpha)$. The identity substitution is written ι. The substitution $[\tau/\alpha]$ maps α to τ and otherwise acts as ι.

Given δ from Γ to Δ, we write the *information increase* relation $\delta : \Gamma \preceq \Delta$ and say Δ *is more informative than* Γ if for all $xD \in \Gamma$, we have $\Delta \vdash \delta[\![xD]\!]$. That is, Δ supports the statements arising from declarations in Γ. We write $\Gamma \preceq \Delta$ if $\iota : \Gamma \preceq \Delta$. If $\delta : \Gamma, \Gamma' \preceq \Theta$ we write $\delta|_\Gamma$ for the restriction of δ to $\mathcal{V}_{\mathrm{TY}}(\Gamma)$.

We write $\delta \equiv \theta : \Gamma \preceq \Delta$ if $\delta : \Gamma \preceq \Delta$, $\theta : \Gamma \preceq \Delta$ and for all $\alpha \in \mathcal{V}_{\mathrm{TY}}(\Gamma)$, $\Delta \vdash \delta\alpha \equiv \theta\alpha$. We will sometimes just write $\delta \equiv \theta$ if the contexts involved are obvious. It is straightforward to verify that $\equiv$ is an equivalence relation for fixed contexts Γ and Δ, and that if $\delta \equiv \theta$ then $\Delta \vdash \delta\tau \equiv \theta\tau$ for any Γ-type τ.

4.1 Stable statements

A statement S is *stable* if information increase preserves it, i.e., if

$$\Gamma \vdash S \quad \text{and} \quad \delta : \Gamma \preceq \Delta \quad \Rightarrow \quad \Delta \vdash \delta S.$$

That is, we can extend a simultaneous substitution on syntax to one on derivations. Since we only consider valid contexts, the statement $\mathbf{valid}$ always holds, is invariant under substitution, hence is stable.

We observe that neutral derivations always ensure stability:

Lemma 1. *If* $\Gamma \Vdash S$ *and* $\delta : \Gamma \preceq \Delta$ *then* $\Delta \vdash \delta S$.

Proof. By induction on derivations. In the case of LOOKUP, it holds by definition of information increase. Otherwise, the proof is by a neutral elimination rule, so the result follows by induction, and admissibility of the corresponding normal elimination rule. □

We have a standard way, effective by construction, to prove stability of most statements: we proceed by induction on derivations. In the NEUTRAL case, stability holds by Lemma 1. Otherwise, we check the non-recursive hypotheses are stable and that recursive hypotheses occur in strictly positive positions, so are stable by induction. In this way we see that $\tau\ \mathbf{type}$ and $\tau \equiv \upsilon$ are stable.

Lemma 2 (Conjunction preserves stability)**.** *If* S *and* S' *are stable then* $S \wedge S'$ *is stable.*

Proof. Suppose S, S' are stable, $\Gamma \vdash S \wedge S'$, and $\delta : \Gamma \preceq \Delta$. In the NEUTRAL case, $\Delta \vdash \delta(S \wedge S')$ by Lemma 1. Otherwise $\Gamma \vdash S$ and $\Gamma \vdash S'$. By stability, $\Delta \vdash \delta S$ and $\Delta \vdash \delta S'$, so $\Delta \vdash \delta(S \wedge S')$. □

We shall exploit the preorder structure of $\preceq$, induced by stability.

Lemma 3. *If* $[\![xD]\!]$ *is stable for every declaration* xD*, then the* $\preceq$ *relation is a preorder, with reflexivity witnessed by the identity substitution* $\iota : \Gamma \preceq \Gamma$*, and transitivity by composition:*

$$\delta : \Gamma \preceq \Delta \text{ and } \theta : \Delta \preceq \Theta \quad \Rightarrow \quad \theta \cdot \delta : \Gamma \preceq \Theta.$$

Proof. Reflexivity follows immediately by applying the LOOKUP and NEUTRAL rules. For transitivity, suppose that $xD \in \Gamma$, then $\Delta \vdash \delta[\![xD]\!]$ since $\delta : \Gamma \preceq \Delta$. Now by stability applied to $\delta[\![xD]\!]$ using θ, we have $\Theta \vdash \theta\delta[\![xD]\!]$ as required. □

5. Constraints: problems at ground mode

We define a *constraint problem* to be a pair of a context Γ and a statement P, where the sanity conditions on the parameters of P hold in Γ, but P itself may not. A *solution* to such a problem is then an information increase $\delta : \Gamma \preceq \Delta$ such that $\Delta \vdash \delta P$. In this setting, the *unification* problem $(\Gamma, \tau \equiv \upsilon)$ stipulates that $\Gamma \vdash \tau\ \mathbf{type} \wedge \upsilon\ \mathbf{type}$, and a solution to the problem (a *unifier*) is given by $\delta : \Gamma \preceq \Delta$ such that $\Delta \vdash \delta\tau \equiv \delta\upsilon$.

We are interested in algorithms to solve problems, preferably in as general a way as possible (that is, by making the smallest information increase necessary to find a solution). For the unification problem, this corresponds to finding a most general unifier. We say the solution $\delta : \Gamma \preceq \Delta$ is *minimal* if, for any other solution $\theta : \Gamma \preceq \Theta$, there exists a substitution $\zeta : \Delta \preceq \Theta$ such that $\theta \equiv \zeta \cdot \delta$ (we say θ *factors through* δ with *cofactor* ζ).

Variables can become more informative either by definition or by substitution. Our algorithms exploit only the former, always choosing solutions of the form $\iota : \Gamma \preceq \Delta$, but we show these minimal with respect to arbitrary information increase. Correspondingly, we write $\Gamma \hat{\preceq} \Delta \vdash P$ to mean that (Γ, P) is a problem with minimal solution $\iota : \Gamma \preceq \Delta$.

Unsurprisingly, stability permits *sound* sequential problem solving:

$$\frac{\iota : \Gamma \preceq \Delta \vdash P \quad \iota : \Delta \preceq \Theta \vdash Q}{\iota : \Gamma \preceq \Theta \vdash P \wedge Q}.$$

If Δ solves P then any more informative context Θ also solves P. More surprisingly, composite problems acquire *minimal* solutions similarly, allowing a 'greedy' strategy.

Lemma 4 (The Optimist's lemma)**.** *The following is admissible:*

$$\frac{\Gamma \hat{\preceq} \Delta \vdash P \quad \Delta \hat{\preceq} \Theta \vdash Q}{\Gamma \hat{\preceq} \Theta \vdash P \wedge Q}.$$

Sketch. Any solution $\phi : \Gamma \preceq \Phi$ to $(\Gamma, P \wedge Q)$ must solve (Γ, P), and hence factor through $\iota : \Gamma \preceq \Delta$. But its cofactor solves (Δ, Q), and hence factors through $\iota : \Delta \preceq \Theta$. For the detailed proof of a more general result, see Lemma 11. □

This sequential approach to problem solving is not the only decomposition justified by stability. McAdam's account of unification [1998] amounts to a concurrent, transactional decomposition of problems. The same context is extended via multiple different substitutions, which are then unified to produce a single substitution.

6. The unification algorithm, formally

We now present the algorithm formally. The structural rule ensures that rigid problems, with $\triangleright$ on each side, decompose into subproblems: by the Optimist's lemma, these we solve sequentially. Otherwise, we have either two variables, or a variable and a type. In each case, we ask how the rightmost type variable in the context helps us, and either solve the problem or continue leftward in the context with an updated constraint. When solving a variable with a type, we must accumulate the type's dependencies as we find them, performing the occurs check to ensure a solution exists.

The rules in Figure 5 define our unification algorithm. The *unify* judgment $\Gamma \twoheadrightarrow \Delta \vdash \tau \equiv \upsilon$ means that given inputs Γ, τ and υ, satisfying the input sanity condition $\Gamma \vdash \tau\ \mathbf{type} \wedge \upsilon\ \mathbf{type}$, unification succeeds, yielding output context Δ.

The *solve* judgment $\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau$ means that given inputs Γ, Ξ, α and τ, solving α with τ succeeds, yielding output context Δ. The idea is that the bar ($\mid$) represents progress in examining context elements in order, and Ξ contains exactly those declarations on which τ depends. Formally, the inputs must satisfy (†):

$\alpha \in \mathcal{V}_{\mathrm{TY}}(\Gamma)$, τ is not a variable,
$\Gamma, \Xi \vdash \tau\ \mathbf{type}$, Ξ contains only type variable declarations
$\beta \in \mathcal{V}_{\mathrm{TY}}(\Xi) \Rightarrow \beta \in FTV(\tau, \Xi)$.

The set $FTV(\tau)$ records those variables occurring free in type τ; the notation extends to (sub-)contexts $FTV(\Xi)$ and composite objects $FTV(\tau, \Xi)$ in the obvious way. Some context entries have no bearing on the problem at hand. We write $x \perp X$ (x is orthogonal to set X of type variables) if x is not a type variable or not in X.

The rules DEFINE and EXPAND have symmetric counterparts, identical apart from interchanging the equated terms in the conclusion. Usually we will ignore these without loss of generality.

$$\boxed{\Gamma \twoheadrightarrow \Delta \vdash \tau \equiv \upsilon}$$

$$\textsc{Decompose}\ \frac{\Gamma \twoheadrightarrow \Delta_0 \vdash \tau_0 \equiv \upsilon_0 \quad \Delta_0 \twoheadrightarrow \Delta \vdash \tau_1 \equiv \upsilon_1}{\Gamma \twoheadrightarrow \Delta \vdash \tau_0 \triangleright \tau_1 \equiv \upsilon_0 \triangleright \upsilon_1}$$

$$\textsc{Idle}\ \frac{}{\Gamma, \alpha D \twoheadrightarrow \Gamma, \alpha D \vdash \alpha \equiv \alpha}$$

$$\textsc{Define}\ \frac{}{\Gamma, \alpha{:=}? \twoheadrightarrow \Gamma, \alpha{:=}\beta \vdash \alpha \equiv \beta}\ \alpha \neq \beta$$

$$\textsc{Ignore}\ \frac{\Gamma \twoheadrightarrow \Delta \vdash \alpha \equiv \beta}{\Gamma, xD \twoheadrightarrow \Delta, xD \vdash \alpha \equiv \beta}\ x \perp \{\alpha, \beta\}$$

$$\textsc{Expand}\ \frac{\Gamma \twoheadrightarrow \Delta \vdash \tau \equiv \beta}{\Gamma, \alpha{:=}\tau \twoheadrightarrow \Delta, \alpha{:=}\tau \vdash \alpha \equiv \beta}\ \alpha \neq \beta$$

$$\textsc{Solve}\ \frac{\Gamma \mid \mathcal{E} \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}{\Gamma \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}\ \tau \text{ not variable}$$

$$\boxed{\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}$$

$$\textsc{Defines}\ \frac{}{\Gamma, \alpha{:=}? \mid \Xi \twoheadrightarrow \Gamma, \Xi, \alpha{:=}\tau \vdash \alpha \equiv \tau}\ \alpha \notin FTV(\tau, \Xi)$$

$$\textsc{Ignores}\ \frac{\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}{\Gamma, xD \mid \Xi \twoheadrightarrow \Delta, xD \vdash \alpha \equiv \tau}\ x \perp FTV(\alpha, \tau, \Xi)$$

$$\textsc{Expands}\ \frac{\Gamma, \Xi \twoheadrightarrow \Delta \vdash \upsilon \equiv \tau}{\Gamma, \alpha{:=}\upsilon \mid \Xi \twoheadrightarrow \Delta, \alpha{:=}\upsilon \vdash \alpha \equiv \tau}\ \alpha \notin FTV(\tau, \Xi)$$

$$\textsc{Depends}\ \frac{\Gamma \mid \beta D, \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}{\Gamma, \beta D \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}\ \alpha \neq \beta, \beta \in FTV(\tau, \Xi)$$

Figure 5. Algorithmic rules for unification

Observe that no rule applies in the case (‡)

$$\Gamma, \alpha D \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau \text{ with } \alpha \in FTV(\tau, \Xi),$$

where the algorithm fails. This is an occurs check failure: α and τ cannot unify if α occurs in τ or in an entry that τ depends on, and τ is not a variable. Given the single type constructor symbol (the function arrow $\triangleright$), there are no failures due to rigid-rigid mismatch. To add these would not significantly complicate matters.

The idea of assertions producing a resulting context goes back at least to Pollack [1990]. Nipkow and Prehofer [1995] use (unordered) input and output contexts to pass information about 'sorts' for Haskell typeclass inference, alongside a conventional substitution-based presentation of unification.

By exposing the contextual structure underlying unification we make termination of the algorithm evident. Each recursive appeal to unification (directly or via the solving process) either shortens the context left of the bar, shortens the overall context, or preserves the context and decomposes types [McBride 2003]. We are correspondingly entitled to reason about the total correctness of unification by induction on the algorithmic rules.

6.1 Soundness and completeness

At present, order in the context is unimportant (providing dependencies are respected) but we will see in Section 8 that the algorithm does keep entries as far right as possible, which will be necessary for generality of type inference.

Lemma 5 (Soundness and generality of unification)**.**

(a) Suppose $\Gamma \twoheadrightarrow \Delta \vdash \tau \equiv \upsilon$. *Then* $\mathcal{V}_{\mathrm{TY}}(\Gamma) = \mathcal{V}_{\mathrm{TY}}(\Delta)$ *and* $\Gamma \mathrel{\hat{\preceq}} \Delta \vdash \tau \equiv \upsilon$.

(b) Suppose $\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau$. *Then* $\mathcal{V}_{\mathrm{TY}}(\Gamma, \Xi) = \mathcal{V}_{\mathrm{TY}}(\Delta)$ *and* $\Gamma, \Xi \mathrel{\hat{\preceq}} \Delta \vdash \alpha \equiv \tau$.

Proof. By induction on the structure of derivations. For each rule, we verify that it preserves the set of type variables and that $\Gamma \preceq \Delta$.

For minimality, it suffices to take some $\theta : \Gamma \preceq \Theta$ such that $\Theta \vdash \theta\tau \equiv \theta\upsilon$, and show $\theta : \Delta \preceq \Theta$. As the type variables of Γ are the same as Δ, we simply note that definitions in Δ hold as equations in Θ for each rule that rewrites or solves the problem.

The only rule not in this form is DECOMPOSE, but solutions to $\tau_0 \triangleright \tau_1 \equiv \upsilon_0 \triangleright \upsilon_1$ are exactly those that solve $\tau_0 \equiv \upsilon_0 \wedge \tau_1 \equiv \upsilon_1$, so it gives a minimal solution by the Optimist's lemma. □

We prove a straightforward lemma about the occurs check, and hence show completeness of unification.

Lemma 6 (Occurs check)**.** *Let* α *be a variable and* τ *a non-variable type such that* $\alpha \in FTV(\tau)$. *There is no context* Θ *and substitution* θ *such that* $\Theta \vdash \theta\alpha \equiv \theta\tau$ *or* $\Theta \vdash \theta\tau \equiv \theta\alpha$.

Proof. Suppose otherwise. Moreover, let Θ contain no definitions (by extending θ to substitute them out). Now, $\theta\alpha \equiv \theta\tau$ ensures $\theta\alpha = \theta\tau$, but as $\alpha \in FTV(\tau)$ and τ is not α, $\theta\tau$ must be a proper subterm of itself, which is impossible. □

Lemma 7 (Completeness of unification)**.** *(a) If* $\theta : \Gamma \preceq \Theta$, $\Gamma \vdash \upsilon\ \mathbf{type} \wedge \tau\ \mathbf{type}$ *and* $\Theta \vdash \theta\upsilon \equiv \theta\tau$, *then there is some context* Δ *such that* $\Gamma \twoheadrightarrow \Delta \vdash \upsilon \equiv \tau$.

(b) Moreover, if $\theta : \Gamma, \Xi \preceq \Theta$ *is such that* $\Theta \vdash \theta\alpha \equiv \theta\tau$ *and the input conditions* (†) *are satisfied, then there is some context* Δ *such that* $\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau$.

Proof. It suffices to show that the algorithm succeeds for every well-formed input in which a solution can exist. As the algorithm terminates, we proceed by induction on its call graph. Each step preserves solutions: if the equation in a conclusion can be solved, so can those in its hypothesis.

The only case the rules omit is the case (‡) where an illegal occurrence of a type variable is rejected. In this case, we are seeking to solve the problem $\alpha \equiv \tau$ in the context $\Gamma, \alpha D \mid \Xi$ and we have $\alpha \in FTV(\tau, \Xi)$. Substituting out the definitions in Ξ from τ, we obtain a type υ such that $\alpha \in FTV(\upsilon)$, υ is not a variable and $\Gamma, \alpha D, \Xi \vdash \upsilon \equiv \tau$. Now the problem $\alpha \equiv \upsilon$ has the same solutions as $\alpha \equiv \tau$, but by Lemma 6, there are no such. □

7. Specifying type inference

We aim to implement type inference for the Hindley-Milner system, so we need to introduce type schemes and the term language. We extend the grammar of statements to express additions to the context (binding statements), well-formed schemes, type assignment and scheme assignment. The final grammar will be:

$$S ::= \mathbf{valid} \mid \tau\ \mathbf{type} \mid \tau \equiv \upsilon \mid S \wedge S \mid xD \succ S \mid \sigma\ \mathbf{scheme} \mid t : \tau \mid s :: \sigma.$$

7.1 Binding statements

To account for schemes and type assignment, we need a controlled way to extend the context. Given statement S and declaration xD, then we define the statement $xD \succ S$, binding x in S, subject to D.

We give a generic introduction rule, but we make use of neutral elimination only for type variables.

$$\frac{\Gamma \vdash \mathbf{ok}_K D \quad \Gamma, yD \vdash [y/x]S}{\Gamma \vdash xD \succ S}\ y \in \mathcal{V}_K \setminus \mathcal{V}_K(\Gamma)$$

$$\frac{\Vdash \alpha D \succ S \quad \vdash [\tau/\alpha]\llbracket \alpha D \rrbracket}{\Vdash [\tau/\alpha]S}\ D \in \mathcal{D}_{\mathrm{TY}}$$

The corresponding normal rule is admissible. If $\Gamma \vdash \alpha D \succ S$ by the introduction rule, then $\Gamma, \beta D \vdash [\beta/\alpha]S$ where β is fresh. But $\Gamma \vdash [\tau/\alpha]\llbracket \alpha D \rrbracket$ implies $\Gamma \vdash [\tau/\beta]\llbracket \beta D \rrbracket$ and hence we can obtain a proof of $\Gamma \vdash [\tau/\alpha]S$ by replacing every appeal to LOOKUP β in the proof of $\Gamma, \beta D \vdash [\beta/\alpha]S$ with the proof of $\Gamma \vdash [\tau/\beta]\llbracket \beta D \rrbracket$. As a consequence, Lemma 1 still holds.

While the introduction rule allows renaming to ensure freshness, in practice we will ignore this and assume that the bound variable name is always fresh for the context.

Lemma 8 (Binding preserves stability). *If xD is a declaration and both $\mathbf{ok}_K D$ and S are stable, then $xD \succ S$ is stable.*

Proof. Suppose S is stable, $\delta : \Gamma \preceq \Delta$, x chosen fresh for Γ and Δ, and $\Gamma \vdash xD \succ S$. In the NEUTRAL case, the result follows by Lemma 1. Otherwise, $\Gamma \vdash \mathbf{ok}_K D$ and $\Gamma, xD \vdash S$. By stability and inductive hypothesis, $\Delta \vdash \delta(\mathbf{ok}_K D)$. Now we have $\delta : \Gamma, xD \preceq \Delta, x(\delta D)$ so we also have $\Delta, x(\delta D) \vdash \delta S$ by stability of S. Hence $\Delta \vdash x(\delta D) \succ \delta S$ and so $\Delta \vdash \delta(xD \succ S)$. □

We extend the binding notation to $\Xi \succ S$, where Ξ is a list of declarations, by: $\mathcal{E} \succ S \mapsto S$ and $(\Xi, xD) \succ S \mapsto \Xi \succ (xD \succ S)$.

If S is a statement and C is a sanity condition for one of its parameters, the statement $xD \succ S$ has sanity condition $xD \succ C$ for the corresponding parameter.

7.2 Type schemes

To handle let-polymorphism, the context must assign type schemes to term variables, rather than monomorphic types. A *type scheme* σ is a type wrapped in one or more $\forall$ quantifiers or $(!\cdot := \cdot \text{ in } \cdot)$ bindings, with the syntax

$$\sigma ::= .\tau \mid \forall \alpha\, \sigma \mid (!\alpha{:=}\tau \text{ in } \sigma).$$

We use explicit definitions in type schemes to avoid the need for substitution in the type inference algorithm.

Schemes arise by discharging a context suffix (a list of type variable declarations) over a type, and any scheme can be viewed in this way. We write $(\Xi \Uparrow \tau)$ for the generalisation of the type τ over the suffix of type variable declarations Ξ, defined by

$$\begin{aligned} \mathcal{E} \Uparrow \tau &\mapsto .\tau \\ \alpha{:=}?, \Xi \Uparrow \tau &\mapsto \forall \alpha\, (\Xi \Uparrow \tau) \\ \alpha{:=}\upsilon, \Xi \Uparrow \tau &\mapsto (!\alpha{:=}\upsilon \text{ in } (\Xi \Uparrow \tau)) \end{aligned}$$

The statement $\sigma\ \mathbf{scheme}$ is then defined by

$$(\Xi \Uparrow \tau)\ \mathbf{scheme} \mapsto \Xi \succ \tau\ \mathbf{type}.$$

The sanity condition is just $\mathbf{valid}$, as for $\tau\ \mathbf{type}$.

7.3 Terms and type assignment

Now we are in a position to reuse the framework already introduced, defining the sort TM, with $\mathcal{V}_{\mathrm{TM}}$ a set of term variables and x ranging over $\mathcal{V}_{\mathrm{TM}}$. Term variable properties $\mathcal{D}_{\mathrm{TM}}$ are scheme assignments of the form $::\sigma$, with $\mathbf{ok}_{\mathrm{TM}}(::\sigma) = \sigma\ \mathbf{scheme}$.

Let s, t, w range over the set of terms with syntax

$$t ::= x \mid t\ t \mid \lambda x.t \mid \text{let } x{:=}t \text{ in } t.$$

The type assignment statement $t : \tau$ is established by the rules in Figure 6. It has two parameters t and τ with sanity conditions $\mathbf{valid}$ and $\tau\ \mathbf{type}$ respectively. We overload notation to define the scheme assignment statement $t :: \sigma$ by

$$t :: (\Xi \Uparrow \tau) \mapsto \Xi \succ t : \tau.$$

Note this gives the parameters t and σ sanity conditions $\mathbf{valid}$ and $\sigma\ \mathbf{scheme}$ as one might expect. This overloading is reasonable because the meaning of $::$ is clear from the context, and the interpretation of declarations embeds them in statements:

$$\llbracket x :: \sigma \rrbracket_{\mathrm{TM}} \mapsto x :: \sigma.$$

$$\boxed{t : \tau}$$

$$\frac{x :: .\upsilon \succ t : \tau}{\lambda x.t : \upsilon \triangleright \tau} \qquad \frac{f : \upsilon \triangleright \tau \quad a : \upsilon}{fa : \tau}$$

$$\frac{s :: \sigma \quad x :: \sigma \succ w : \tau}{\text{let } x{:=}s \text{ in } w : \tau} \qquad \frac{t : \tau \quad \tau \equiv \upsilon}{t : \upsilon}$$

Figure 6. Declarative rules for type assignment

The definition of $\Gamma \preceq \Delta$ requires Δ to assign a term variable all the types that Γ assigns it, but allows x to become more polymorphic and acquire new types. This notion certainly retains stability: every variable lookup can be simulated in the more general context. However, it allows arbitrary generalisation of the schemes assigned to term variables which are incompatible with the known and intended value of those variables.

As Wells [2002] points out, HM type inference is not in this respect compositional. He carefully distinguishes principal *typings*, given the right to demand more polymorphism, from Milner's principal *type schemes* and analyses how the language of types must be extended to express principal typings.

We, too, note this distinction. We cannot hope to find principal types with respect to $\preceq$, so we will define a subrelation $\sqsubseteq$ to capture Milner's compromise, requiring that, for $\delta : \Gamma \preceq \Delta$,

$$x :: \sigma \in \Gamma \Rightarrow x :: \delta\sigma \in \Delta.$$

If $\Gamma \sqsubseteq \Delta$, then Δ assigns the *same* type schemes to term variables as Γ does (modulo substitution). Since the unification algorithm ignores term variables, it must preserve this property. This is not the full story, however; we need to extend the notion of context to complete the definition of the $\sqsubseteq$ relation.

8. Generalising *local* type variables

We have previously observed, but not yet exploited, the importance of declaration order in the context, and that we move declarations left as little as possible. Thus rightmost entries are those most local to the problem we are solving. This will be useful when we come to implement type inference for the 'let' construct, as we want to generalise over 'local' type variables but not 'global' variables.

In order to keep track of locality in the context, we need another kind of context entry: the $\S$ separator. We add a new validity rule

$$\frac{\Gamma \vdash \mathbf{valid}}{\Gamma \S \vdash \mathbf{valid}}.$$

We must then refine the $\sqsubseteq$ relation to respect these $\S$ divisions. Let $\lfloor$ be the partial function from contexts Γ and natural numbers n which truncates Γ after n $\S$ separators, provided Γ contains at least n such:

$$\begin{array}{rcl} \Xi \lfloor 0 & \mapsto & \Xi \\ \Xi \S \Gamma \lfloor 0 & \mapsto & \Xi \\ \Xi \S \Gamma \lfloor n+1 & \mapsto & \Xi \S (\Gamma \lfloor n) \\ \Xi \lfloor n+1 & \text{undefined} & \end{array}$$

We write $\delta : \Gamma \sqsubseteq \Delta$ if δ is a substitution from Γ to Δ such that, for all $xD \in \Gamma \lfloor n$, we have that $\Delta \lfloor n$ is defined, $\Delta \lfloor n \vdash \delta[\![xD]\!]$ and

$$x :: \sigma \in \Gamma \;\Rightarrow\; x :: \delta\sigma \in \Delta.$$

We thus make the $\S$-separated sections of Γ and Δ correspond, so that declarations in the first n sections of Γ can be interpreted over the first n sections of Δ. As a consequence, 'moving left of $\S$' is an irrevocable commitment. In particular, we note that

$$\iota : \Gamma \S \alpha{:=}?, \Delta \sqsubseteq \Gamma, \alpha{:=}? \S \Delta \quad \text{but} \quad \iota : \Gamma, \alpha{:=}? \S \Delta \not\sqsubseteq \Gamma \S \alpha{:=}?, \Delta$$

Note also that if $\delta : \Gamma \S \Gamma' \sqsubseteq \Delta \S \Delta'$, where Γ and Δ contain the same number of $\S$ separators, then $\delta|_\Gamma : \Gamma \sqsubseteq \Delta$.

When the contexts contain only type variables, the two relations $\preceq$ and $\sqsubseteq$ coincide; the latter is a proper subrelation if the contexts also contain term variables. Hence, most of the previous results hold if we replace $\preceq$ with $\sqsubseteq$ throughout.

8.1 Amending the unification algorithm

Replacing $\preceq$ with $\sqsubseteq$ makes extra work only in the unification algorithm, because it acts structurally on contexts, which may now contain $\S$ separators. We complete the algorithmic rules:

$$\textsc{Skip}\ \frac{\Gamma \twoheadrightarrow \Delta \vdash \alpha \equiv \beta}{\Gamma \S \twoheadrightarrow \Delta \S \vdash \alpha \equiv \beta}$$

$$\textsc{Repossess}\ \frac{\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau}{\Gamma \S \mid \Xi \twoheadrightarrow \Delta \S \vdash \alpha \equiv \tau}$$

We must correspondingly update the induction in Lemma 5 to show that adding the new rules preserves soundness and generality. For the SKIP rule, correctness follows immediately from this lemma:

Lemma 9. *If* $\Gamma \mathbin{\hat{\sqsubseteq}} \Delta \vdash S$ *then* $\Gamma \S \mathbin{\hat{\sqsubseteq}} \Delta \S \vdash S$.

Proof. If $\Gamma \sqsubseteq \Delta$ then $\Gamma \S \sqsubseteq \Delta \S$ by definition. If $\Delta \vdash S$ then $\Delta \S \vdash S$ since the LOOKUP rule is the only one that extracts information from the context, and it ignores the $\S$.

Now let $\theta : \Gamma \S \sqsubseteq \Theta \S \Xi$ be such that $\Theta \S \Xi \vdash S$. By definition of $\sqsubseteq$, we must have $\theta : \Gamma \sqsubseteq \Theta$, so by minimality there exists $\zeta : \Delta \sqsubseteq \Theta$ with $\theta \equiv \zeta \cdot \iota$. Then $\zeta : \Delta \S \sqsubseteq \Theta \S \Xi$ and we are done. □

The REPOSSESS rule is so named because it moves declarations in Ξ to the left of the $\S$ separator, thereby 'repossessing' them. To guarantee a solution most general with respect to $\sqsubseteq$, we show that Ξ's leftward journey is really necessary.

Lemma 10 (Soundness and generality of the REPOSSESS rule)**.** *Suppose* $\Gamma \S \mid \Xi \twoheadrightarrow \Delta \S \vdash \alpha \equiv \tau$. *Then* $\mathcal{V}_{\mathrm{TY}}(\Gamma \S \Xi) = \mathcal{V}_{\mathrm{TY}}(\Delta \S)$ *and* $\Gamma \S \Xi \sqsubseteq \Delta \S \vdash \alpha \equiv \tau$.

Proof. We extend the structural induction in Lemma 5 with an extra case. The only proof of $\Gamma \S \mid \Xi \twoheadrightarrow \Delta \S \vdash \alpha \equiv \tau$ is by REPOSSESS, so inversion gives $\Gamma \mid \Xi \twoheadrightarrow \Delta \vdash \alpha \equiv \tau$. By induction, $\mathcal{V}_{\mathrm{TY}}(\Gamma, \Xi) = \mathcal{V}_{\mathrm{TY}}(\Delta)$ and $\Gamma, \Xi \sqsubseteq \Delta \vdash \alpha \equiv \tau$.

We immediately observe that $\Gamma \S \Xi \sqsubseteq \Delta \S$, $\Delta \S \vdash \alpha \equiv \tau$ and

$$\mathcal{V}_{\mathrm{TY}}(\Gamma \S \Xi) = \mathcal{V}_{\mathrm{TY}}(\Gamma, \Xi) = \mathcal{V}_{\mathrm{TY}}(\Delta) = \mathcal{V}_{\mathrm{TY}}(\Delta \S).$$

For minimality, suppose $\theta : \Gamma \S \Xi \sqsubseteq \Theta \S \Phi$ and $\Theta \S \Phi \vdash \theta\alpha \equiv \theta\tau$. Observe that $\alpha \in \mathcal{V}_{\mathrm{TY}}(\Gamma)$ and $\beta \in \mathcal{V}_{\mathrm{TY}}(\Xi) \Rightarrow \beta \in FTV(\tau, \Xi)$ by the conditions for the algorithmic judgment. Now $\theta\alpha$ is a Θ-type and $\theta\tau$ is equal to it, so the only declarations in Φ that $\theta\tau$ (hereditarily) depends on must be definitions over Θ. But all the variables declared in Ξ are used in τ, so there is a substitution $\psi : \Gamma \S \Xi \sqsubseteq \Theta \S$ that agrees with θ on Γ and maps variables in Ξ to their definitions in Θ.

Hence $\psi : \Gamma, \Xi \sqsubseteq \Theta$ and $\Theta \vdash \psi\alpha \equiv \psi\tau$, so by hypothesis there exists $\zeta : \Delta \sqsubseteq \Theta$ such that $\psi \equiv \zeta \cdot \iota : \Gamma, \Xi \sqsubseteq \Theta$. Note that $\psi \equiv \theta : \Gamma \S \Xi \sqsubseteq \Theta \S \Phi$. Then $\zeta : \Delta \S \sqsubseteq \Theta \S \Phi$ and $\psi \equiv \zeta \cdot \iota : \Gamma \S \Xi \sqsubseteq \Theta \S \Phi$, so $\theta \equiv \zeta \cdot \iota : \Gamma \S \Xi \sqsubseteq \Theta \S \Phi$. □

9. Type inference problems and their solutions

Type inference involves making the statement $t : \tau$ hold, but unlike unification, the type should be an *output* of problem-solving along with the solution context. We need a more liberal definition than that of constraint problems. We associate a *mode* with each parameter in a statement: either 'input' or 'output'. For simplicity, assume statements always have one parameter of each mode (which may be trivial or composite). We now extend the apparatus of minimal solutions to problems with outputs.

What can outputs be, and how can we compare them? An *output set* is a set B closed under substitution, such that every context Γ induces a preorder $\Gamma \vdash \cdot \subset \cdot$ on B which is congruent with respect to the definitional equality, i.e. if $\Gamma \vdash \alpha \equiv \tau \wedge \beta \equiv \upsilon$, then $\Gamma \vdash b \subset c$ if and only if $\Gamma \vdash [\tau/\alpha]b \subset [\upsilon/\beta]c$. This is easily verified for each preorder we use.

We need subsequent problems to depend on the results of earlier problems, threading the output from one into the input of the next. Thus we must index problems to determine the input parameters.

Let A be an output set. An *A-indexed problem family Q for B* is an output set B and a family of input parameters for a statement, indexed by elements of A, such that the *simplicity condition* holds: for all $a, a' \in A$, contexts Γ and output parameter values $b \in B$,

$$\Gamma \vdash a \subset a' \;\wedge\; \Gamma \vdash Q[a']\, b \quad \Rightarrow \quad \Gamma \vdash Q[a]\, b.$$

We write $Q[a]\, b$ for the statement with input at index a and output value b, and $Q[a]$ for the sanity conditions on the input parameters at index a. We use $\Gamma \vdash \cdot \subset_Q \cdot$ for the preorder on the output set. The idea behind this contravariant condition is that the preorder represents specialisation of solutions, so if a problem can be solved with an input a' then it can be solved with the more general a.

Now we can generalise the notion of constraint problem and its solution. An *inference problem* consists of a context Γ, an A-indexed problem family Q and an index $a \in A$ such that $\Gamma \vdash Q[a]$.

A *solution* of it consists of an information increase $\delta : \Gamma \sqsubseteq \Delta$ and a value for the output parameter $b \in B$ such that $\Delta \vdash (\delta(Q[a]))\, b$.

The preorder on outputs induces a preorder on context-output pairs, with $\delta : (\Gamma, a) \sqsubseteq (\Delta, b)$ if $\delta : \Gamma \sqsubseteq \Delta$ and $\Delta \vdash \delta a \subset b$. We will look for minimal solutions with respect to this preorder, and write $\Gamma \circ Q[a] \hat{\sqsubseteq} \Delta \bullet b$ if $(\iota : \Gamma \sqsubseteq \Delta, b)$ is a solution (i.e. $\Delta \vdash Q[a]\, b$) and for all solutions $(\theta : \Gamma \sqsubseteq \Theta, c)$ we have $\zeta : (\Delta, b) \sqsubseteq (\Theta, c)$ for some ζ such that $\theta \equiv \zeta \cdot \iota$. As with unification, we only use the identity substitution but are minimal with respect to any solution.

A *problem P for B* is a problem family indexed by the unit set with the trivial preorder. We simply omit the index in this case.

9.1 The Optimist's lemma

Let P be a problem for A and let Q be an A-indexed family for B. Then the conjunction ΣPQ is a problem for $A \times B$ with statement

$$(\Sigma PQ)(a, b) \;\mapsto\; Pa \wedge Q[a]\, b$$

and the preorder defined pointwise. This 'dependent' generalisation of $P \wedge Q$ allows the output of P to be threaded into Q. The Optimist's lemma correspondingly generalises:

Lemma 11 (The Optimist's lemma for inference problems)**.**

$$\frac{\Gamma \circ P \hat{\sqsubseteq} \Delta \bullet b \quad \Delta \circ Q[b] \hat{\sqsubseteq} \Theta \bullet c}{\Gamma \circ (\Sigma PQ) \hat{\sqsubseteq} \Theta \bullet (b, c)}.$$

Proof. Since $\Gamma \sqsubseteq \Delta$ and $\Delta \sqsubseteq \Theta$, we have $\Gamma \sqsubseteq \Theta$ by (updating) Lemma 3. Furthermore, $\Theta \vdash (\Sigma PQ)(b, c)$ since $\Theta \vdash Q[b]\, c$ by assumption and $\Delta \vdash Pb$ so stability gives $\Theta \vdash Pb$.

For minimality, suppose there is a solution $(\phi : \Gamma \sqsubseteq \Phi, (b', c'))$, so $\Phi \vdash (\phi P) b'$ and $\Phi \vdash (\phi Q)[b']\, c'$. Since $\Gamma \circ P \hat{\sqsubseteq} \Delta \bullet b$, there exists $\zeta : \Delta \sqsubseteq \Phi$ with $\Phi \vdash \zeta b \subset b'$ and $\phi \equiv \zeta \cdot \iota$. By the simplicity condition, $\Phi \vdash (\phi Q)[\zeta b]\, c'$ and hence $\Phi \vdash (\zeta(Q[b]))\, c'$. But $\Delta \circ Q[b] \hat{\sqsubseteq} \Theta \bullet c$, so there exists $\xi : \Theta \sqsubseteq \Phi$ such that $\Phi \vdash \xi c \subset c'$ and $\zeta \equiv \xi \cdot \iota$. Hence $\Phi \vdash \xi(b, c) \subset (b', c')$ so $\xi : (\Theta, (b, c)) \sqsubseteq (\Phi, (b', c'))$, and $\phi \equiv \zeta \cdot \iota \equiv (\xi \cdot \iota) \cdot \iota \equiv \xi \cdot \iota$. □

9.2 The Generalist's lemma

We have considered problems with abstract inputs and outputs, but which concrete values do we actually use? We want to solve type inference problems, so we are interested in types and type schemes.

The statement $t :: \sigma$ defines a problem for the set of schemes with preorder given by $\Gamma \vdash (\Xi \Uparrow \tau) \subset_{::} (\Psi \Uparrow \upsilon)$ if there is some $\psi : \Gamma \fatsemi \Xi \sqsubseteq \Gamma \fatsemi \Psi$ such that $\Gamma \fatsemi \Psi \vdash \psi\tau \equiv \upsilon$ and $\psi|_\Gamma \equiv \iota$. That is, $\Gamma \vdash \sigma \subset_{::} \sigma'$ if σ is a more general type scheme than σ'.

Since types are just schemes with no quantifiers, we instantiate the above definition with $\Xi = \mathcal{E} = \Psi$, to get a preorder on types: $\Gamma \vdash \tau \subset_: \upsilon$ if $\Gamma \vdash \tau \equiv \upsilon$.

Thus the type inference problem is given by a context Γ and a term parameter t as input to the type assignment statement. Following the definitions, a solution is an information increase $\delta : \Gamma \sqsubseteq \Delta$ and a type τ such that $\Delta \vdash \tau\ \mathbf{type} \wedge t : \tau$. A solution with output τ is minimal if, given any other solution, we can find a substitution that unifies τ and the other type: that is, τ is a principal type.

In the type inference algorithm, we will use $\fatsemi$ to determine what can be generalised, based on the following lemma.

Lemma 12 (The Generalist's lemma)**.** *This rule is admissible:*

$$\frac{(\Gamma \fatsemi) \circ (t :) \hat{\sqsubseteq} (\Delta \fatsemi \Xi) \bullet \tau}{\Gamma \circ (t ::) \hat{\sqsubseteq} \Delta \bullet (\Xi \Uparrow \tau)}.$$

Proof. If $\Gamma \fatsemi \sqsubseteq \Delta \fatsemi \Xi$ then $\Gamma \sqsubseteq \Delta$ by definition. Furthermore, $\Delta \vdash t :: (\Xi \Uparrow \tau)$ is defined to be $\Delta \vdash \Xi \succ t : \tau$, which holds iff $\Delta \fatsemi \Xi \vdash t : \tau$.

For minimality, suppose $\theta : \Gamma \sqsubseteq \Theta$ is an information increase and $(\Psi \Uparrow \upsilon)$ is a scheme such that $\Theta \vdash t :: (\Psi \Uparrow \upsilon)$. Then $\Theta, \Psi \vdash t : \upsilon$. Now $\theta : \Gamma \fatsemi \sqsubseteq \Theta \fatsemi \Psi$ and $\Theta \fatsemi \Psi \vdash t : \upsilon$, so by minimality of the hypothesis there is a substitution $\zeta : \Delta \fatsemi \Xi \sqsubseteq \Theta \fatsemi \Psi$ such that $\theta \equiv \zeta \cdot \iota$ and $\Theta \fatsemi \Psi \vdash \zeta\tau \equiv \upsilon$. Then by definition $\zeta|_\Delta : (\Delta, (\Xi \Uparrow \tau)) \sqsubseteq (\Theta, (\Psi \Uparrow \upsilon))$ and $\theta \equiv \zeta|_\Delta \cdot \iota : \Gamma \sqsubseteq \Theta$. □

9.3 The binding lemmas

Just as we have a general notion of conjunction problems, so we can regard binding statements as problems. There are two ways to do so, depending on the mode of the bound property. Each has a corresponding minimality result.

First, if Q is a problem for A, then $x :: \sigma \succ Q$ is also a problem for A where we regard σ as an input. It has statement

$$(x :: \sigma \succ Q)a \;\mapsto\; x :: \sigma \succ Qa$$

and preorder given by $\Gamma \vdash a \subset_{(x :: \sigma \succ Q)} b$ if $\Gamma, x :: \sigma \vdash a \subset_Q b$. Minimal solutions are found by bringing x into scope temporarily.

Lemma 13. *If Ξ does not contain any $\fatsemi$ separators, then we have:*

$$\frac{(\Gamma, x :: \sigma) \circ Q \hat{\sqsubseteq} (\Delta, x :: \sigma, \Xi) \bullet a}{\Gamma \circ (x :: \sigma \succ Q) \hat{\sqsubseteq} (\Delta, \Xi) \bullet a}.$$

Proof. If $\Gamma, x :: \sigma \sqsubseteq \Delta, x :: \sigma, \Xi$ then $\Gamma \sqsubseteq \Delta, \Xi$ since nothing in Ξ can depend on x. If $\Delta, x :: \sigma, \Xi \vdash Qa$ then $\Delta, \Xi, x :: \sigma \vdash Qa$ (permuting the context) and hence $\Delta, \Xi \vdash x :: \sigma \succ Qa$.

If $\theta : \Gamma \sqsubseteq \Theta$ is such that $\Theta \vdash x :: \theta\sigma \succ (\theta Q)a'$, then by inversion, $\Theta, x :: \theta\sigma \vdash (\theta Q)a'$. By minimality of the hypothesis, there is $\zeta : \Delta, x :: \sigma, \Xi \sqsubseteq \Theta, x :: \theta\sigma$ such that $\Theta, x :: \theta\sigma \vdash \theta a \subset_Q a'$ and $\theta \equiv \zeta \cdot \iota$. Hence $\zeta : \Delta, \Xi \sqsubseteq \Theta$ and $\Theta \vdash \theta a \subset_{(x :: \sigma \succ Q)} a'$. □

Alternatively, we can regard a type variable binding as being initially unknown, and obtain the problem $\alpha \succ Q$ whose output is a pair of a type and a value in A. The corresponding statement is

$$(\alpha \succ Q)(\tau, b) \;\mapsto\; [\tau/\alpha](Qb)$$

and the output preorder is given by $\Gamma \vdash (\tau, a) \subset_{(\alpha \succ Q)} (\upsilon, b)$ if $\Gamma \vdash \tau \equiv \upsilon$ and $\Gamma \vdash [\tau/\alpha]a \subset_Q [\upsilon/\alpha]b$. Minimal solutions arise by adding an unknown to the context and returning it as the output:

Lemma 14.

$$\frac{(\Gamma, \alpha :=?) \circ Q \hat{\sqsubseteq} \Delta \bullet b}{\Gamma \circ (\alpha \succ Q) \hat{\sqsubseteq} \Delta \bullet (\alpha, b)}$$

Proof. By hypothesis, $\Delta \vdash Qb$ so clearly $\Delta \vdash [\alpha/\alpha](Qb)$. Moreover, $\Gamma, \alpha :=? \sqsubseteq \Delta$ so $\Gamma \sqsubseteq \Delta$. If $\theta : \Gamma \sqsubseteq \Theta$ is such that $\Theta \vdash [\upsilon/\alpha]((\theta Q)c)$, then $\Theta \vdash ([\upsilon/\alpha]\theta Q)([\upsilon/\alpha]c)$. By minimality of the hypothesis with the substitution $[\upsilon/\alpha] \cdot \theta : \Gamma, \alpha :=? \sqsubseteq \Theta$, there is some $\zeta : \Delta \sqsubseteq \Theta$ such that $\Theta \vdash \zeta b \subset_Q ([\upsilon/\alpha]c)$ and $[\upsilon/\alpha] \cdot \theta \equiv \zeta \cdot \iota$. Hence $\zeta : (\Delta, (\alpha, b)) \sqsubseteq (\Theta, (\upsilon, c))$. □

9.4 Transforming type assignment into type inference

To transform a rule into an algorithmic form, we proceed clockwise starting from the conclusion. For each hypothesis, we must ensure that the problem is fully specified, inserting variables to stand for unknown problem inputs. Moreover, we cannot pattern match on problem outputs, so we ensure there are schematic variables in output positions, fixing things up with appeals to unification.

Figure 7 shows the transformed version of the declarative rule system. The λ-rule now binds a fresh name for the argument type, which gets replaced with an unknown in the algorithm. The rule for application assigns types to the function and argument separately, then inserts an equation with a fresh name for the codomain type.

$$\boxed{t : \tau}$$

$$\frac{\beta{:=}v, x :: .\beta \succ t : \tau}{\lambda x.t : v \triangleright \tau} \qquad \frac{f : \chi \quad a : v \quad \beta{:=}\tau \succ \chi \equiv v \triangleright \beta}{fa : \tau}$$

$$\frac{s :: \sigma \quad x :: \sigma \succ w : \tau}{\text{let } x{:=}s \text{ in } w : \tau} \qquad \frac{t : \tau \quad \tau \equiv v}{t : v}$$

Figure 7. Transformed rules for type assignment

We must verify that the rule systems in Figures 6 and 7 are equivalent. This is mostly straightforward, as fresh name bindings can be substituted out. The only difficulty is in the application rule, where an equation is introduced. If an application has a type in the old system, it can be assigned the same type in the new system with using a reflexive equation. Conversely, if an application has a type in the new system, then using the conversion with the equation allows the same type to be assigned in the old system.

Given the transformed rules, we construct the algorithm to match. We establish the type inference assertion $\Gamma \circ (t :) \twoheadrightarrow \Delta \bullet \tau$ and the scheme inference assertion $\Gamma \circ (s ::) \twoheadrightarrow \Delta \bullet \sigma$ by the rules in Figure 8. As they are structural on terms, they yield a terminating algorithm, and hence the implementation in Subsection 9.6. The Optimist's lemma permits sequential solution of problems and the binding lemmas let us interpret binding statements as problems.

$$\boxed{\Gamma \circ (s ::) \twoheadrightarrow \Delta \bullet \sigma}$$

$$\textsc{Gen}\ \frac{(\Gamma, \Xi) \circ (s :) \twoheadrightarrow (\Delta, \Xi) \bullet v}{\Gamma \circ (s ::) \twoheadrightarrow \Delta \bullet (\Xi \Uparrow v)}$$

$$\boxed{\Gamma \circ (t :) \twoheadrightarrow \Delta \bullet \tau}$$

$$\textsc{Var}\ \frac{x :: (\Xi \Uparrow v) \in \Gamma}{\Gamma \circ (x :) \twoheadrightarrow (\Gamma, \Xi) \bullet v}$$

$$\textsc{Abs}\ \frac{(\Gamma, \alpha{:=}?, x :: .\alpha) \circ (w :) \twoheadrightarrow (\Delta, x :: .\alpha, \Xi) \bullet v}{\Gamma \circ (\lambda x.w :) \twoheadrightarrow (\Delta, \Xi) \bullet (\alpha \triangleright v)}\ \alpha \notin \mathcal{V}_{\text{TY}}(\Gamma)$$

$$\textsc{App}\ \frac{\Gamma \circ (f :) \twoheadrightarrow \Delta_0 \bullet \chi \quad \Delta_0 \circ (a :) \twoheadrightarrow \Delta_1 \bullet v \quad \Delta_1, \beta{:=}? \twoheadrightarrow \Delta \vdash \chi \equiv v \triangleright \beta}{\Gamma \circ (fa :) \twoheadrightarrow \Delta \bullet \beta}\ \beta \notin \mathcal{V}_{\text{TY}}(\Delta_1)$$

$$\textsc{Let}\ \frac{\Gamma \circ (s ::) \twoheadrightarrow \Delta_0 \bullet \sigma \quad (\Delta_0, x :: \sigma) \circ (w :) \twoheadrightarrow (\Delta, x :: \sigma, \Xi) \bullet \chi}{\Gamma \circ (\text{let } x{:=}s \text{ in } w :) \twoheadrightarrow (\Delta, \Xi) \bullet \chi}$$

Figure 8. Algorithmic rules for type inference

9.5 Soundness and completeness

Since the algorithmic rules correspond directly to the transformed declarative system in Figure 7, we can easily prove soundness, completeness and generality of type inference with respect to this system. Each proof is by induction on derivations, observing that each algorithmic rule maintains the appropriate properties.

Recall that a type inference problem (Γ, P) has statement $t : \tau$ where t is a term and τ is the output type. A scheme inference problem has statement $t :: \sigma$ where σ is the output scheme.

Lemma 15 (Soundness of type inference)**.** *If (Γ, P) is a type or scheme inference problem, and $\Gamma \circ P \twoheadrightarrow \Delta \bullet a$, then $\Gamma \sqsubseteq \Delta$ and $\Delta \vdash Pa$.*

Proof. We maintain this property as an invariant in all the rules. □

To prove generality, we use the admissible rules in the Optimist's, Generalist's and binding lemmas. The algorithmic rules map to compositions of these, with multiple hypotheses corresponding to conjunctions of problems. To apply the Optimist's lemma, we must check that the problem on the right satisfies the 'simplicity condition'. For LET, this means we need

$$\Gamma \vdash \sigma \subset_{::} \sigma' \ \wedge\ \Gamma, x :: \sigma' \vdash w : \chi \quad \Rightarrow \quad \Gamma, x :: \sigma \vdash w : \chi,$$

which says that if a solution can be found with x having a given type scheme then one can be found with it having a more general scheme. The APP case is even more straightforward.

Lemma 16 (Generality of type inference)**.** *If (Γ, P) is a type or scheme inference problem, and $\Gamma \circ P \twoheadrightarrow \Delta \bullet a$, then $\Gamma \circ P \hat{\sqsubseteq} \Delta \bullet a$.*

Proof. Given soundness (Lemma 15), it remains to show generality, i.e. that each algorithmic rule becomes admissible in the transformed declarative system if we replace $\twoheadrightarrow$ with $\hat{\sqsubseteq}$.

For the VAR rule, suppose $\theta : \Gamma \sqsubseteq \Theta$ and $\Theta \vdash x : \tau$. By inversion, the proof must consist of the LOOKUP rule followed by eliminating $\Theta \Vdash x :: (\theta\Xi \Uparrow \theta v)$ with some Θ-types. Hence it determines a map from the unbound type variables of Ξ to types over Θ, i.e. a substitution $\zeta : \Gamma, \Xi \sqsubseteq \Theta$ that agrees with θ on Γ and maps type variables in Ξ to their definitions in Θ.

All the remaining cases are covered by the previous lemmas. The Generalist's lemma proves exactly the property required for the GEN rule. The ABS rule is minimal by Lemmas 13 and 14. The APP rule is minimal by two uses of the Optimist's lemma, Lemma 14 and minimality of unification. The LET rule is minimal by the Optimist's lemma and Lemma 13. □

Lemma 17 (Completeness of type inference)**.** *If (Γ, P) is a type or scheme inference problem, and there exist $\theta : \Gamma \sqsubseteq \Theta$ and a' such that $\Theta \vdash (\theta P)a'$, then $\Gamma \circ P \twoheadrightarrow \Delta \bullet a$ for some context Δ and output a.*

Proof. We proceed by induction on the derivation of $\Theta \vdash (\theta P)a'$. Every case in the transformed declarative system (excluding the conversion rule) is covered by the algorithm, and it reduces the problem to an equivalent form, thereby preserving solutions. Thus if a solution exists, then the algorithm will succeed. □

9.6 Implementation of type inference

Figure 9 shows the Haskell implementation of our type inference algorithm. Note that the monadic `fail` is called if scope checking fails, whereas `error` signals violation of an algorithmic invariant.

Figure 9(a) implements type schemes. It is convenient to represent bound variables by de Bruijn indices and free variables (in the context) by names [McBride and McKinna 2004b]. We use Haskell's type system to prevent some incorrect manipulations of indices by defining a 'successor' type `Index`, where the outermost bound variable is represented by `Z` and other variables are wrapped in the `S` constructor [Bellegarde and Hook 1994; Bird and Paterson 1999].

```
data Index a = Z | S a deriving (Functor, Foldable)
data Schm a = Type (Ty a)
            | All (Schm (Index a))
            | LetS (Ty a) (Schm (Index a))
  deriving (Functor, Foldable)
type Scheme = Schm TyName
```

(a) Type schemes

```
specialise :: Scheme → Contextual Type
specialise (Type τ) = return τ
specialise σ = do
    let (d, σ') = unpack σ
    β ← fresh d
    specialise (fmap (fromS β) σ')
  where
    unpack :: Scheme → (TyDecl, Schm (Index TyName))
    unpack (All σ')      = (? , σ')
    unpack (LetS τ σ') = (!τ, σ')
    fromS :: TyName → Index TyName → TyName
    fromS β Z      = β
    fromS β (S α) = α
```

(b) Specialisation

```
bind :: TyName → Scheme → Schm (Index TyName)
bind α = fmap help
  where
    help :: TyName → Index TyName
    help β | α ≡ β     = Z
           | otherwise = S β

(⇑) :: Suffix → Type → Scheme
ℰ              ⇑ τ = Type τ
(α:=?  :> Ξ) ⇑ τ = All (bind α (Ξ ⇑ τ))
(α:=! υ :> Ξ) ⇑ τ = LetS υ (bind α (Ξ ⇑ τ))

generaliseOver :: Contextual Type → Contextual Scheme
generaliseOver mt = do
    modifyContext (:<⨾)
    τ ← mt
    Ξ ← skimContext ℰ
    return (Ξ ⇑ τ)
  where
    skimContext :: Suffix → Contextual Suffix
    skimContext Ξ = do
      Γ :< vD ← getContext
      putContext Γ
      case vD of
        ⨾      → return Ξ
        TY αD → skimContext (αD :> Ξ)
        TM _  → error "Unexpected TM variable!"
```

(c) Generalisation

```
data Tm a = X a
          | Tm a :$ Tm a
          | Lam a (Tm a)
          | Let a (Tm a) (Tm a)
  deriving (Functor, Foldable)

type TmName = String
type Term   = Tm TmName

data TmEntry = TmName :: Scheme
data Entry   = TY TyEntry | TM TmEntry | ⨾

find :: TmName → Contextual Scheme
find x = getContext >>= help
  where
    help :: Context → Contextual Scheme
    help (Γ :< TM (y :: σ)) | x ≡ y = return σ
    help (Γ :< _)                   = help Γ
    help ℰ                          = fail "Missing var!"
```

(d) Terms and context entries

```
(≻) :: TmEntry → Contextual a → Contextual a
x :: σ ≻ ma = do
    modifyContext (:<TM (x :: σ))
    a ← ma
    modifyContext extract
    return a
  where
    extract :: Context → Context
    extract (Γ :< TM (y :: _)) | x ≡ y = Γ
    extract (Γ :< TY xD) = (extract Γ) :< TY xD
    extract (Γ :< _) = error "Bad context entry!"
    extract ℰ        = error "Missing TM variable!"
```

(e) Bringing term variables into scope

```
infer :: Term → Contextual Type
infer (X x) = find x >>= specialise
infer (Lam x w) = do
  α ← fresh ?
  υ ← x :: Type (V α) ≻ infer w
  return (V α ▷ υ)
infer (f :$ a) = do
  χ ← infer f
  υ ← infer a
  β ← fresh ?
  unify χ (υ ▷ V β)
  return (V β)
infer (Let x s w) = do
  σ ← generaliseOver (infer s)
  x :: σ ≻ infer w
```

(f) Type inference

Figure 9. Haskell implementation of type inference

Figures 9(b) and 9(c) implement specialisation and generalisation of type schemes. The former unpacks a scheme with fresh names; the latter 'skims' entries off the top of the context to the $\S$ marker.

Figure 9(d) implements the data type of terms, and gives the final definition of Entry including type and term variable declarations and $\S$ markers. It implements the find function to look up a term variable in the context and return its scheme.

Figure 9(e) implements the $(\succ)$ operator to evaluate Contextual code in the scope of a term variable, then remove it afterwards. This is necessary for dealing with λ-abstractions and let-bindings.

Finally, Figure 9(f) implements the type inference algorithm itself. It proceeds structurally over the term, following the rules in Figure 8 and using the monadic operations.

10. Discussion

We have arrived at an implementation of Hindley-Milner type inference which involves all the same steps as Algorithm $\mathcal{W}$, but not necessarily in the same order. In particular, the dependency panic which seizes $\mathcal{W}$ in the let-rule here becomes an invariant that the underlying unification algorithm maintain a well-founded context.

Our algorithm is presented as a problem transformation system locally preserving all possible solutions, hence finding a most general global solution if any at all. Accumulating solutions to decomposed problems is justified simply by stability of solutions on information increase. We have established a discipline of problem solving, happily complete for Hindley-Milner type inference, but in any case coupling soundness with generality.

Maintain context validity, make definitions anywhere and only where there is no choice, so the solutions you find will be general and generalisable locally: this is a key design principle for elaboration of high-level code in systems like Epigram and Agda; bugs arise from its transgression. Our disciplined account of 'current information' in terms of contexts and their information ordering provides a principled means to investigate and repair these troubles.

We are, however, missing yet more context. Our task was greatly simplified by studying a structural type inference process for 'finished' expressions in a setting where unification is complete. Each subproblem is either solved or rejected on first inspection—there is never a need for a 'later, perhaps' outcome. As a result, 'direct style' recursive programming is adequate to the task. If problems could get stuck, how might we abandon them and return to them later? By storing their *context*, of course!

Here, we have combined the *linguistic* contexts for various sorts of variable; our next acquisition is the *syntactic* context of the target term, interspersing variable declarations with pieces of its *zipper* [Huet 1997]. We thus enable a flexible traversal strategy, refocusing wherever progress can be made. The tree-like proof states of McBride's thesis evolved into exactly such 'zippers with binding' in the implementation of Epigram.

As we have seen, 'information increase' is really the elaboration of simultaneous substitution from variables-and-terms to declarations-and-derivations. Our analysis of role declaration plays in derivation shows that stability is endemic—an action of hereditary substitution on 'cut-free' derivations. And that is just what it should be. We have rationalised Hindley-Milner type inference, adapting a discipline for incremental term construction in dependent types to manage unknowns for incremental problem solving. The analysis can only become clearer, the technology simpler, as we identify these two kinds of construction, mediating *problems as types*.

References

F. Baader and W. Snyder. Unification theory. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

F. Bellegarde and J. Hook. Substitution: a formal methods case study using monads and transformations. *Sci. Comp. Programming*, 23(2-3):287–311, 1994.

R. Bird and R. Paterson. de Bruijn notation as a nested datatype. *J. Functional Programming*, 9(1):77–91, 1999.

D. Clément, T. Despeyroux, G. Kahn, and J. Despeyroux. A simple applicative language: mini-ML. In *Proc. LISP and Functional Programming*, pages 13–27. ACM, 1986.

L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. POPL '82*, pages 207–212. ACM, 1982.

J. Dunfield. Greedy bidirectional polymorphism. In *Proc. ML '09*, pages 15–26. ACM, 2009.

GHC Team. The GHC user's guide, version 6.12.1. Section 7.5. Extensions to the "deriving" mechanism, 2009.

G. Huet. The Zipper. *J. Functional Programming*, 7(5):549–554, 1997.

B. J. McAdam. On the unification of substitutions in type inference. In *Proc. IFL' 98*, pages 139–154. Springer, 1998.

C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

C. McBride. First-Order Unification by Structural Recursion. *J. Functional Programming*, 13(6), 2003.

C. McBride and J. McKinna. The view from the left. *J. Functional Programming*, 14(1):69–111, 2004a.

C. McBride and J. McKinna. Functional pearl: I am not a number–I am a free variable. In *Proc. Haskell workshop*, pages 1–9. ACM, 2004b.

D. Miller. Unification under a mixed prefix. *J. Symbolic Computation*, 14 (4):321–358, 1992.

R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17(3):348–375, 1978.

W. Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *J. Automated Reasoning*, 23(3):299–318, 1999.

T. Nipkow and C. Prehofer. Type reconstruction for type classes. *J. Functional Programming*, 5(2):201–224, 1995.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

R. Pollack. Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks*, 1990.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965.

J. B. Wells. The essence of principal typings. In *Proc. ICALP '02*, pages 913–925. Springer, 2002.